CS 3100 – Models of Computation – Fall 2011
This assignment is worth 8% of the total points for assignments
100 points total

September 2, 2011

Assignment 2, Posted on: 8/26 Due: 9/8 midnight

Powerset.py (**20 points**) Write a python function to compute the powerset of a given set or list (the function should work for both; hint: do a list(S) inside the function). Return a **list of lists**.

Examples:

```
pow({'ab', 'bc'}) --> [['ab', 'bc'], ['bc'], ['ab'], []]

pow(['ab', 'bc']) --> [['ab', 'bc'], ['bc'], ['ab'], []]
```

Write the output to file `Powerset.out`. Test it on at least the following inputs in the given order.

The powerset of a set S is the set of all subsets of S. Think of a recursive way to compute this. Here's a hint: Suppose `Ps` is the powerset of `[1,2,3]`. and `Ps1` the powerset of `[2,3]` (obtained by removing `1` from `[1,2,3]`. How do we produce `Ps` from `Ps1`? Think of the following questions, and you will see how to proceed:

- Is `Ps1` a subset of `Ps`?
- What else is there in `Ps`?

We will be running `Powerset.py` from outside, so arrange it to be a script. Once invoked, it must run and output into the file `Powerset.out` the powerset of the following sets/lists. You should print each powerset using `print` (which by default puts a new line after each item is printed).

Inputs:

- `set()`
- `set('')`
- `[]`
- `['']`
- `['a']`
- `['a', 'b', 'c', 'd', 'e']`

What to submit (recap): The file Powerset.py (containing the code to compute powerset, and with also Python calls that call the powerset function with the requisite inputs, and the required `__main__` line to run it as a script).

MkDFA.py (**20 points**) Define a function `mk_dfa` whose definition is sketched below. A DFA is represented using a dict of the form:

```
{"Q":Q, "Sigma":Sigma, "Delta":Delta, "q0":q0, "F":F})
```

Here, `Q` is a **non-empty** set of strings (state names), `Sigma` is a set of **non-empty single-character strings** (alphabet), `q0` is a **state belonging to `Q`**, and `F` is a **possibly non-empty set of states, and is also a subset of `Q`**. `Delta` is a **total** function represented as a hash-table, mapping a pair (q, c) (**where `q` in `Q`** and `c` **in `Sigma`)** to a new state $q1$ where `q1` **is also in `Q`**.

Implement all the checks in **boldface** font given above as `asserts` in Python. Test that all the checks are working. Submit this terminal session of the checks happening as file `MkDFATests.txt`.

```
def mk_dfa(Q, Sigma, Delta, q0, F):
    """Make a DFA with the given traits. Delta is supplied as a hash-map (dict).
    """
    # Do all the checks listed in boldface fonts, above, using Python asserts.
    #
    # If all OK, return DFA as a dict
    return({"Q":Q, "Sigma":Sigma, "Delta":Delta, "q0":q0, "F":F})
```

What to submit: A file containing the code for `MkDFA.py` that we will run from outside (details are similar to those mentioned already several times). When so run, the following commands must be executed:

```
Q1 = {’S0’,’S1’}

Sigma1 = {’a’,’b’}

Delta1 = {(’S0’, ’a’): ’S0’, (’S1’, ’a’): ’S0’, (’S1’, ’b’): ’S1’, (’S0’, ’b’): ’S1’}

q01 = ’S0’

F1 = {’S1’}

mk_dfa(Q1,Sigma1,Delta1,q01,F1)
```

We will (or our grading script will) basically look for the final `mk_dfa` call printing the correct DFA `dict` object on the console, and see if it is correct.

DotDFA.py (**20 points**) Describe all the functions in the file `DotDFA.py` that I wrote for your use in this class. Your descriptions should be in the form of a reasonable help string documenting each function. Use multiple lines to document each function. You should aim to be clear, yet succinct. I've provided some help strings in some functions - not very detailed. You should simply remove my help string and/or reuse it, but write a nice help string yourself.

```
# Contents of file DotDFA.py

def homos(S,f):
    """String homomorphism wrt lambda f
       homos("abcd",hm) --> ’bcde’  where hm = lambda x: chr( (ord(x)+1) % 256 )
    """
    return "".join(map(f,S))

def dotsan_map(x):
    """Students have to think and conclude whether this is a homomorphism or not!
    """
    if x in {  "{",   " ",   "’",   "}" }:
        return ""
    elif x == ",":
        return "_"
    else:
        return x

def dot_san_str(S):
    """Make dot like strings which are in set of states notation.
    """
    return homos(S, dotsan_map)

def prDotHeader(fl):
    print (r’digraph G {’, file=fl)
    print (r’/* Defaults */’, file=fl)
    print (r’  fontsize = 12;’, file=fl)
    print (r’  ratio = compress; ’, file=fl)
    print (r’  rankdir=LR; ’, file=fl)
    print (r’/* Bounding box */’, file=fl)
    print (r’   size = "4,4";’, file=fl)

def prNonFinalNodeName(fl, q):
    print (dot_san_str(q), r’[shape=circle, peripheries=1];’, file=fl)

def prFinalNodeName(fl, q):
    # Could write like print (q, r’[shape=circle, peripheries=2];’, file=fl)
    # But am documenting use of trailing comma to suppress \n . In Python3 we supply end = ’’
    print(dot_san_str(q), file=fl, end=’’) # end with no CR
    print(r’ [shape=circle, peripheries=2];’, file=fl) # end with a CR

def prOrientation(fl):
    print(r’/* Orientation */’, file=fl)
    print(r’orientation = landscape;’, file=fl)

def prEdges_w_bh(fl, D):
    print(r’/* The graph itself */’, file=fl)
    print(r’""  -> ’, dot_san_str(D["q0"]), ";", file=fl)
    for QcQ in D["Delta"].items():
        print(dot_san_str(QcQ[0][0]), r’ -> ’,
             dot_san_str(QcQ[1]), r’[label="’, dot_san_str(QcQ[0][1]), r’"];’, file=fl)

def prEdges(fl, D):
    """Suppress BH.
    """
    print(r’/* The graph itself */’, file=fl)
    print(r’""  -> ’, dot_san_str(D["q0"]), ";", file=fl)
    for QcQ in D["Delta"].items():
        if (((QcQ[0][0]) != "BH") & (QcQ[1] != "BH")):
            print(dot_san_str(QcQ[0][0]), r’ -> ’,
                 dot_san_str(QcQ[1]), r’[label="’, dot_san_str(QcQ[0][1]), r’"];’, file=fl)

def prClosing(fl):
    print(r’/* Unix command: dot -Tps exdfa.dot >! exdfa.ps */’, file=fl)
    print(r"/* For further details, see the ‘dot’ manual    */", file=fl)
    print(r"}", file=fl)

def prNodeDefs_w_bh(fl, D):
    print(r’/* Node definitions */’, file=fl)
    print(r’  "" [shape=plaintext];’, file=fl)  # Start state arrow is from "" to I
    # All non-accepts are single circles
    for q in D["Q"] - D["F"]:
        prNonFinalNodeName(fl, q)
    for q in D["F"]:
        prFinalNodeName(fl, q)

def prNodeDefs(fl, D):
    """Suppress BH.
    """
    print(r’/* Node definitions */’, file=fl)
    print(r’  "" [shape=plaintext];’, file=fl)  # Start state arrow is from "" to I
    # All non-accepts are single circles
    for q in D["Q"] - D["F"]:
        if (q != "BH"):
            prNonFinalNodeName(fl, q)
    for q in D["F"]:
        prFinalNodeName(fl, q)

def dot_dfa(D, fname):
    """Generate a dot file with the automaton in it. Run the dot file through
    dot and generate a ps file.
    """
    fl = open(fname, ’w’)
    #-- digraph decl
    prDotHeader(fl)
    #-- node names and how to draw them
    prNodeDefs(fl, D)
    #-- orientation - now landscape
    prOrientation(fl)
    #-- edges
    prEdges(fl, D)
    #-- closing
    prClosing(fl)
```

What to submit:

- The result of feeding the call `mk_dfa(Q1,Sigma1,Delta1,q01,F1)` to function `dot_dfa` giving file name `DotDFA.dot`. We will look for `DotDFA.dot` being written out, grading that coming out successfully.
- Next, execute the commands `help(homos)`, `help(dotsan_map)`, etc. (in sequence for all the functions). These `help` calls must be executed when we call `DotDFA.py`. We will look for the help strings coming on to the console and grade them.

- You should run the command `dot -Tps DotDFA.dot > DotDFA.ps`, convert the `.ps` file to a `.pdf` file, and submit the PDF as `DotDFA.pdf` (Hint: you can use `ps2pdf13 file.ps` to generate `file.pdf`.) In fact, you should accomplish this function from within the `DotDFA.py` invocation itself. Hint: You can execute Unix commands from your Python file using `os.system("unix command presented as a string")`.
- We will look at your code, grade it, and also your `DotDFA.pdf`.

DFA0101.py (**30 points**)

Design (on paper) a DFA that accepts all strings over $\Sigma = \{0, 1\}$ that end in 0101. Now type in suitable Python commands to create this DFA using the call `mk_dfa(Q1,Sigma1,Delta1,q01,F1)` to function `dot_dfa` giving filename `DFA0101.dot`. Produce a `pdf` as described above.

We will look at your code, grade it, and also grade your final `pdf`.

Pascal.txt (**5 points**) Describe the function `pascal` on Page 8 of notes2.pdf in some detail (a page), explaining how all the functions involved work.

Numeric.txt (**5 points**) Describe the function `nthnumeric` on Page 14 of notes2.pdf in some detail (a page), explaining how all the calculations and homomorphisms involved work.

Extra Stuff: Here is a script that may help TAs grade your assignments. I enclose it to help you learn more Python.

```
#!/Library/Frameworks/Python.framework/Versions/Current/bin/python3

import os
import os.path

def read_file_of_pgm_files():
    filename = "FilesToRun" # default first
    while True:
        if os.path.exists(filename):
            print(’Found file ’ + filename)
            break
        else:
            print(’Did not find file ’ + filename + ’. Try again’)
            filename = raw_input(’Enter filename: ’ )
    return filename

def gradeit(asg):
    fhandle = open(read_file_of_pgm_files())
    # Get all pgm names in fhandle, and pick out all .py files thru filter
    pgm_files = list(filter(lambda fil: fil[-3::] == ".py", fhandle.read().split(’\n’)))
    # print("Program files = ", pgm_files)
    # Get all user names
    users = os.listdir(asg)
    print("Users = ", users)
    for user in users:
        print("Grading user " + user)
        for pgm in pgm_files:
            print("Gonna run " + pgm + " of user " + user)
            os.system(asg+"/"+user+"/"+pgm)

if __name__ == "__main__":
```

```
gradeit("asg1")
```